Unlink <modname> Usage : Unlinks module(s) from memory @WCREATE Syntax: Wcreate [opt] or /wX [-s=type] xpos ypos xsiz ysiz fcol bcol [bord] Usage : Initialize and create windows Opts : -? = display help -z = read command lines from stdin -s=type = set screen type for a window on a new screen @XMODE Syntax: XMode <devname> [params] Usage : Displays or changes theparameters of an SCF type device @COCOPR Syntax: cocopr [<opts>] {<path> [<opts>]} Function: display file in specified format gets defaults from /dd/sys/env.file Options : -c set columns per page -f use form feed for trailer -h=num set number of lines after header

AUSTRALIAN OS9 NEWSLETTER

EDITOR	**Gordon Bentzen**	**(07) 344-3881**
SUB-EDITOR	**Bob Devries**	**(07) 372-7816**
TREASURER	**Don Berrie**	**(07) 375-1284**
LIBRARIAN	**Jean-Pierre Jacquet**	**(07) 372-4675**
	Fax Messages	**(07) 372-8325**
SUPPORT	**Brisbane OS9 Users Group**	

-l=num set line length -m=num set left margin -n=num set starting line number and incr -o truncate lines longer than lnlen -p=num set number of lines per page -t=num number of lines in trailer -u do not use title -u=title use specfied title -x=num set starting page number -z[=path] read file names from stdin or <path> if given @CONTROL Syntax: control [-e] Usage : Control Panel to set palettes, mouse and keyboard parameters and monitor type for Multi-Vue. Selectable from desk utilities menu as the Control Panel. Opts : -e = execute the environment file @GCLOCK

ADDRESSES

Editorial Material:
Gordon Bentzen
8 Odin Street
SUNNYBANK Qld 4109

Library Requests:
Jean-Pierre Jacquet
27 Hampton Street
DURACK Qld 4077

CONTENTS

Volume 6 | October 1992 | Number 9

AUSTRALIAN OS9 NEWSLETTER
Newsletter of the National OS9 User Group
Volume 6 Number 9

EDITOR : Gordon Bentzen
SUBEDITOR : Bob Devries
TREASURER : Don Berrie
LIBRARIAN : Jean-Pierre Jacquet
SUPPORT : Brisbane OS9 Level 2 Users Group.

This OS-9 Machine just won't die! I refer of course to the humble CoCo3 with its Level 2 version of 6809 OS-9 which is still in use by the majority of our members. And a National CoCoFest in Melbourne later this month even yet!

It is great to see this continued support for OS-9 and the National OS-9 Usergroup for yet another year. Our membership for the new subscription year has already reached 46 which I must say has somewhat surprised us in spite of an active International OS-9 community.

As we have suggested several times before, the future of OS-9 for personal users will rest with OSK and the 68xxx based machines. We are anxiously awaiting an article from Don Berrie on his recently acquired MM/1 and from what I have observed of Don's demos, the MM/1 runs just like a CoCo but MUCH, MUCH faster.

So for those OSK'ers who have supported us for such a long time, we are certain to have more of interest for you in the pages of future newsletters. So what about an article from you to get things rolling?

It is probably about time that I got serious again about our appeal for newsletter articles, so how about each of you take the time to send us something to share with others. Submissions to the newsletter can take the form of a tutorial, hints and tips, questions or problems or even some of those frustrations. In fact anything at all to do with OS-9 would be of interest to others.

HARD DRIVES ARE GREAT

I would like to share with you one of my own little frustrations which involves Hard Drives on my CoCo3. I have been running a 20meg Hard Drive and Burke & Burke XT interface for some time now and wonder how I ever managed without it. As always happens though, no matter how much storage space, RAM or whatever one has, it inevitably seems to decrease to "not enough". Now one would think that having a reliable system that has given very little trouble over the past twelve months the wise thing would be to leave it alone.
Well, as we are not always driven by wisdom, I entered into a programme of significant system improvement. A good friend of mine had generously offered to me a spare RLL controller card which would suit the Burke & Burke interface and allow my /H0 20 meg drive to be formatted to 30 meg, and my /H1 10 meg drive to be formatted to 15 meg. A total of 45 meg Hard Disk storage instead of 30 meg. This just had to be a worthwhile upgrade, so let's get into it.
THE PLAN Now the first thing is to make a "backup" of /H0. That took just sixteen 80 track 720k disks, not bad for what was there, and another ten 720k disks for the data on /H1. I used the Shareware programme "Stream" which was mentioned in last month's editorial. (Stream.ar is available from our librarian upon request)
Next, new drive descriptors are needed for /H0 and /H1. The utility supplied by Burke & Burke, DDMAKER is used to make the required descriptors. Then build a new OS9Boot which will include the new descriptors. Now switch off the power and remove the old MFM controller card from the B & B interface, fit the RLL controller and put everything back together. Well done!
Switch on and "boot" OS-9 from floppy drive /D0. So far so good. Make sure pause is not active on the Term window to be used, tmode -pause. Now the command, Format /H0. Several cups of coffee later, a "free /H0" command reports 120 thousand odd free sectors, 30meg, Great!!
Next, "Bootport" OS9Boot from the floppy to /H0, and then use "Stream" to restore all the files to the hard drive. Some time later this process is completed and the "acid test" is will it still "boot" from /H0.

YOU GUESSED IT ! CoCo does not want to play anymore, OS-9 just won't "boot", and then it does. After several attempts it appears that the No-Go's far outweigh the Go's. The OS9Boot is re-installed, checked, built again etc, until, with time running short and the newsletter due out, a decision needs to be made. So everything is undone and the MFM controller is again in charge of a 20meg /h0 and 10meg /h1.

Where else could one get such entertainment?

Cheers, Gordon.

A C Tutorial
Chapter 10 - File Input/Output

OUTPUT TO A FILE

Load and display the file named FORMOUT.C for your first example of writing data to a file. We begin as before with the "include" statement for "stdio.h", then define some variables for use in the example including a rather strange looking new type. The type "FILE" is used for a file variable and is defined in the "stdio.h" file. It is used to define a file pointer for use in file operations. The definition of C contains the requirement for a pointer to a "FILE", and as usual, the name can be any valid variable name.

OPENING A FILE

Before we can write to a file, we must open it. What this really means is that we must tell the system that we want to write to a file and what the filename is. We do this with the "fopen" function illustrated in the first line of the program. The file pointer, "fp" in our case, points to the file and two arguments are required in the parentheses, the filename first, followed by the file type. The filename is any valid DOS filename, and can be expressed in upper or lower case letters, or even mixed if you so desire. It is enclosed in double quotes. For this example we have chosen the name TENLINES.TXT. This file should not exist on your disk at this time. If you have a file with this name, you should change its name or move it because when we execute this program, its contents will be erased. If you don't have a file by this name, that is good because we will create one and put some data into it.

READING ("r")

The second parameter is the file attribute and can be any of three letters, "r", "w", or "a", and must be lower case. When an "r" is used, the file is opened for reading, a "w" is used to indicate a file to be used for writing, and an "a" indicates that you desire to append additional data to the data already in an existing file. Opening a file for reading requires that the file already exist. If it does not exist, the file pointer will be set to NULL and can be checked by the program.

WRITING ("w")

When a file is opened for writing, it will be created if it does not already exist and it will be reset if it does resulting in deletion of any data already there.

APPENDING ("a")

When a file is opened for appending, it will be created if it does not already exist and it will be initially empty. If it does exist, the data input point will be the end of the present data so that any new data will be added to any data that already exists in the file.

OUTPUTTING TO THE FILE

The job of actually outputting to the file is nearly identical to the outputting we have already done to the standard output device. The only real differences are the new function names and the addition of the file pointer as one of the function arguments. In the example program, "fprintf" replaces our familiar "printf" function name, and the file pointer defined earlier is the first argument within the parentheses. The remainder of the statement looks like, and in fact is identical to, the "printf" statement.

CLOSING A FILE

To close a file, you simply use the function "fclose" with the file pointer in the parentheses. Actually, in this simple program, it is not necessary to close the file because the system will close all open files before returning to DOS. It would be good programming practice for you to get in the habit of closing all files in spite of the fact that they will be closed automatically, because that would act as a reminder to you of what files are open at the end of each program. You can open a file for writing, close it, and reopen it for reading, then close it, and open it again for appending, etc. Each time you open it, you could use the same file pointer, or you could use a different one. The file pointer is simply a tool that you use to point to a file and you decide what file it will point to. Compile and run this program. When you run it, you will not get any output to the monitor because it doesn't generate any. After running it, look at your directory for a file named TENLINES.TXT and "type" it. That is where your output will be. Compare the output with that specified in the program. It should agree. Do not erase the file named TENLINES.TXT yet. We will use it in some of the other examples in this chapter.

OUTPUTTING A SINGLE CHARACTER AT A TIME

Load the next example file, CHAROUT.C, and display it on your monitor. This program will illustrate how

to output a single character at a time. The program begins with the 'include' statement, then defines some variables including a file pointer. We have called the file pointer 'point' this time, but we could have used any other valid variable name. We then define a string of characters to use in the output function using a 'strcpy' function. We are ready to open the file for appending and we do so in the 'fopen' function, except this time we use the lower cases for the filename. This is done simply to illustrate that DOS doesn't care about the case of the filename. Notice that the file will be opened for appending so we will add to the lines inserted during the last program. The program is actually two nested 'for' loops. The outer loop is simply a count to ten so that we will go through the inner loop ten times. The inner loop calls the function 'putc' repeatedly until a character in 'others' is detected to be a zero.

THE 'putc' FUNCTION

The part of the program we are interested in is the 'putc' function. It outputs one character at a time, the character being the first argument in the parentheses and the file pointer being the second and last argument. Why the designer of C made the pointer first in the 'fprintf' function, and last in the 'putc' function is a good question for which there may be no answer. It seems like this would have been a good place to have used some consistency. When the textline 'others' is exhausted, a newline is needed because a newline was not included in the definition above. A single 'putc' is then executed which outputs the '\n' character to return the carriage and do a linefeed. When the outer loop has been executed ten times, the program closes the file and terminates. Compile and run this program but once again there will be no output to the monitor. Following execution of the program, 'type' the file named TENLINES.TXT and you will see that the 10 new lines were added to the end of the 10 that already existed. If you run it again, yet another 10 lines will be added. Once again, do not erase this file because we are still not finished with it.

READING A FILE

Load the file named READCHAR.C and display it on your monitor. This is our first program to read a file. This program begins with the familiar 'include', some data definitions, and the file opening statement which should require no explanation except for the fact that an 'r' is used here because we want to read it. In this program, we check to see that the file exists, and if it does, we execute the main body of the program. If it doesn't, we print a message and quit. If the file does not exist, the system will set the pointer equal to NULL which we can test. The main body of the program is one 'do while' loop in which a single character is read from the file and output to the monitor until an EOF (end of file) is detected from the input file. The file is then closed and the program is terminated.

CAUTION CAUTION CAUTION

At this point, we have the potential for one of the most common and most perplexing problems of programming in C. The variable returned from the 'getc' function is a character, so we could use a 'char' variable for this purpose. There is a problem with that however, because on some, if not most, implementations of C, the EOF returns a minus one which a 'char' type variable is not capable of containing. A 'char' type variable can only have the values of zero to 255, so it will return a 255 for a minus one on those compilers that use a minus one for EOF. This is a very frustrating problem to try to find because no diagnostic is given. The program simply can never find the EOF and will therefore never terminate the loop. This is easy to prevent, always use an 'int' type variable for use in returning an EOF. You can tell what your compiler uses for EOF by looking at the 'stdio.h' file where EOF is defined. That is the standard place to define such values. There is another problem with this program but we will worry about it when we get to the next program and solve it with the one following that. After you compile and run this program and are satisfied with the results, it would be a good exercise to change the name of 'TENLINES.TXT' and run the program again to see that the NULL test actually works as stated. Be sure to change the name back because we are still not finished with 'TENLINES.TXT'.

READING A WORD AT A TIME

Load and display the file named READTEXT.C for an example of how to read a word at a time. This program is nearly identical as the last except that this program uses the 'fscanf' function to read in a string at a time. Because the 'fscanf' function stops reading when it finds a space or a newline character, it will read a word at a time, and display the results one word to a line. You will see this when you compile and run it, but first we must examine a programming problem.

THIS IS A PROBLEM

Inspection of the program will reveal that when we read data in and detect the EOF, we print out something before we check for the EOF resulting in an extra line of printout. What we usually print out is

the same thing printed on the prior pass through the loop because it is still in the buffer 'oneword'. We therefore must check for EOF before we execute the 'printf' function. This has been done in READGOOD.C, which you will shortly examine, compile, and execute. Compile and execute the original program we have been studying, READTEXT.C and observe the output. If you haven't changed TENLINES.TXT you will end up with 'Additional' and 'lines.' on two separate lines with an extra 'lines.' displayed because of the 'printf' before checking for EOF. Compile and execute READGOOD.C and observe that the extra 'lines.' does not get displayed because of the extra check for the EOF in the middle of the loop. This was also the problem referred to when we looked at READCHAR.C, but I chose not to expound on it there because the error in the output was not so obvious.

FINALLY, WE READ A FULL LINE

Load and display the file READLINE.C for an example of reading a complete line. This program is very similar to those we have been studying except for the addition of a new quantity, the NULL. We are using 'fgets' which reads in an entire line, including the newline character into a buffer. The buffer to be read into is the first argument in the function call, and the maximum number of characters to read is the second argument, followed by the file pointer. This function will read characters into the input buffer until it either finds a newline character, or it reads the maximum number of characters allowed minus one. It leaves one character for the end of string NULL character. In addition, if it finds an EOF, it will return a value of NULL. In our example, when the EOF is found, the pointer 'c' will be assigned the value of NULL. NULL is defined as zero in your 'stdio.h' file. When we find that 'c' has been assigned the value of NULL, we can stop processing data, but we must check before we print just like in the last program. Last of course, we close the file.

HOW TO USE A VARIABLE FILENAME

Load and display the file ANYFILE.C for an example of reading from any file. This program asks the user for the filename desired, reads in the filename and opens that file for reading. The entire file is then read and displayed on the monitor. It should pose no problems to your understanding so no additional comments will be made. Compile and run this program. When it requests a filename, enter the name and extension of any text file available, even one of the example C programs.

HOW DO WE PRINT?

Load the last example file in this chapter, the one named PRINTDAT.C for an example of how to print. This program should not present any surprises to you so we will move very quickly through it. Once again, we open TENLINES.TXT for reading and we open PRN for writing. Printing is identical to writing data to a disk file except that we use a standard name for the filename. There are no definite standards as far as the name or names to be used for the printer, but some of the usual names are, 'PRN', 'LPT', 'LPT1', and 'LPT2'. Check your documentation for your particular implementation. Some of the newest compilers use a predefined file pointer such as 'stdprn' for the print file. Once again, check your documentation. The program is simply a loop in which a character is read, and if it is not the EOF, it is displayed and printed. When the EOF is found, the input file and the printer output files are both closed. You can now erase TENLINES.TXT from your disk. We will not be using it in any of the later chapters.

PROGRAMMING EXERCISES

1. Write a program that will prompt for a filename for a read file, prompt for a filename for a write file, and open both plus a file to the printer. Enter a loop that will read a character, and output it to the file, the printer, and the monitor. Stop at EOF.

2. Prompt for a filename to read. Read the file a line at a time and display it on the monitor with line numbers.

oooooooooo0000000000oooooooooo

File Allocation Table Display
by Chris Bergerson
edited by Bob Devries

Here is one of the programmes I have recently received through my affiliation with the OS9 Community Network, a group of OS9 users who communicate regularly via the FIDO BBS network. I have made one modification to the programme, commented appropriately, to clear the garbage from the display where the graphics went beyond the actual FAT table boundaries. Previously, the author just left in the map what data was read even though it did not represent the FAT data. Here now is the Basic09 programme, with the document file after, as produced by Chris Bergerson, and included in the archive.

Bob Devries

```
PROCEDURE fat_display
 0000      (* Procedure to graphically display a disk's File Allocation Table
 0042      (* C. Bergerson 7/31/88   OS9 Level II
 0068
 0069      TYPE registers=cc,a,b,dp:BYTE; x,y,u:INTEGER
 008E      DIM regs:registers
 0097      DIM dpath:BYTE
 009E      DIM spath:INTEGER
 00A5      DIM name:STRING[3]
 00B1      DIM i,dd_map,pages,count:INTEGER
 00C4      DIM gr:INTEGER
 00CB      DIM map(4960):BYTE \(* Approximately 10 meg at a time.
 00F9       \ \ \ \ \ \ \ \ \(* (4960*8*256=10,158,080)
 011D      DIM z:INTEGER \(* Added by Bob Devries
 013B
 013C
 013D      RUN syscall($0C,regs) \(* Get process, user IDs
 0163      gr=regs.a \(* Group #= Process ID
 0184      SHELL "merge /dd/sys/stdfonts;display 1b 3a c8 01"
 01B2      OPEN #spath,"/w"
 01BD      RUN gfx2(spath,"DWSet",5,0,0,80,10,1,0,2)
 01E7      RUN gfx2(spath,"Select")
 01FA      PRINT #spath,"Graphic File Allocation Table Display..."
 022B      INPUT #spath,"Disk name to display, (e.g. /H0): ",name
 025A      OPEN #dpath,name+"@":READ
 026A      SEEK #dpath,4 \(* Go to LSN0, byte 4 for dd_map
 0293      GET #dpath,dd_map \(* Get size of FAT
 02AF      pages=INT((dd_map-1)/4960)+1 \(* Do about 10 meg at a time
 02E0      SEEK #dpath,256 \(* Go to start of FAT
 02FF
 0300      FOR count=1 TO pages
 0311        (* RUN gfx2(spath,"clear")
 032B        GET #dpath,map
 0335        IF count=pages THEN  \(* Added by Bob Devries
 0359          FOR z=dd_map-(pages-1)*4960 TO 4960 \(* Added by Bob Devries
 038D            map(z)=255 \(* Added by Bob Devries
 03AF          NEXT z \(* Added by Bob Devries
 03D1        ENDIF  \(* Added by Bob Devries
 03EA        RUN gfx2(spath,"GPLoad",gr,1,5,639,61,4960)
 0413        PUT #spath,map
 041D        RUN gfx2(spath,"Bell")
 042E        RUN gfx2(spath,"Put",gr,1,0,0)
 044C        RUN gfx2(spath,"CurXY",0,8)
 0464        PRINT #spath,"page "; count; " of "; pages; "   ";
 0488        RUN gfx2(spath,"CurXY",20,8)
```

```
04A0        INPUT #spath,"Press <enter> to continue",name
04C6      NEXT count
04D1
04D2      RUN gfx2("KillBuff",gr,1)
04EA      RUN gfx2(1,"Select")
04FB      RUN gfx2(spath,"DWEnd")
050D      CLOSE #dpath
0513      CLOSE #spath
```

Just what we needed... another utility for graphically displaying a disk's free space!

This program will create the required type 5 window, and then ask for the name of the disk. It works quite well for hard disks. For floppies, only the first few lines will be meaningful, since I assume that the File Allocation Table represents at least 10 meg.

Blue, (or black on a monocrome monitor), represents used clusters. White means that the clusters are free. Each pixel is one cluster.

Runb and Gfx2 must be in memory or in your current execution directory! Program also will attempt to load /dd/sys/stdfonts, so you better be sure they're there. After unARing, be sure to set the e attribute, and move to your execution directory. Call by simply typing:

fat_display

I find this util kind of interesting for seeing how OS9 fills up the disk.

Questions/comments are welcome.

Chris Bergerson
CIS 72227,127

oooooooooo0000000000ooooooooooo

Getting the OS9 Attitude

(This is a limited peek into a powerful, complex system. It is not intended as a "how to" manual, rather to help you "get the feel". Some specifics are given and every attempt has been made to be accurate in these cases.)

From the very outset I will make it clear that, as it comes, OS9 for CoCo is a rather crippled system because of its packaging and lack of support from the producer. There are several things you can do, as soon as you are able, that will make it vastly superior to what it is in its original form.

First, have Level 2 on a CoCo3. There is no comparison between a 32 or 40 column screen and an 80 column screen, but for strange reasons OS9 arrives in a low resolution package and on single sided, 35 track disks. Personally, I wouldn't even consider using it that way. It is like having a powerful new car with special full time brakes installed and all but one window painted black.

So you have an 80 column computer; how do you get OS9 to use it? There is an article by RICKULAND entitled "The First Step" included in this collection. It tells you how to switch to 80 columns, double sided 40 track disks and high speed drive stepping. It is a fairly long tedious procedure, but when it's done, you'll be able to see what you're doing and you'll have more than double the disk space.

Another real aid in using OS9 is in the System Modules database. It is written by OS9 programmer Kevin Darling and is called SCF EDITOR PLUS - LEVEL TWO. Again it is a task and a half to install, but is WELL worth it. It is a command line editor that makes life in the OS9 world of long command lines greatly easier, especially if you're not an expert typist. If you do not have a friend who can install this feature for you the guys on the SIG will help you through it like they did me. I will talk more about this later.

There are many other improvements and additions to OS9 that you will get as you go along, but these are tops in my opinion. Next in line would be the new shell, SHELL+, with several improvements. As of this writing the highest version is v2.1. The sooner you can arrive at an updated system the better off you'll be.

Coming to OS9 from CoCo Disk Extended Color BASIC involves something besides the software and dreams of owning a hard drive - if you don't already have one. No doubt you've heard about the power and versatility

of OS9. It is, indeed, several steps ahead of DECB in many respects. But the change in environments can be almost overwhelming if you don't grasp a few essentials - a few of the differences between DECB and OS9. You can't maintain a DECB attitude while learning OS9.

DECB is a BASIC language with some of the power of a DOS (disk operating system) thrown in. OS9 is an operating system with some of the power of a language thrown in. This much is mundane. But getting into the actual differences is both interesting and essential.

A simple concept that sheds light on the move to OS9 can be illustrated this way. If you had all the money you wanted, but all audio/video systems were alike, it would take you minutes to buy a complete system. But with all the hundreds or thousands of TVs, CDs, amps, tuners, speakers, stereos, tape players, recorders, multi media systems, etc., etc., in the equation, it is a major job deciding on an intelligent system purchase.

The more options an operating system has, the more decisions you must make, the more you must remember, and the more occasions you have to make mistakes. What might involve 3 factors in a simple system might involve 10 in a more powerful system. The 3 would give you 9 paths to choose from. The 10 would give you 100 choices - for starters.

The importance of this point can't be over estimated. And it directly relates to one of the things that causes the most trouble and consternation to new OS9 users - multiple directory disk organization. That will be our starting point and our prime consideration.

For all practical purposes an OS9 system might as well have 6 or 8 (or more) different drives. You can imagine how much you could do by organizing them all in creative order! But you can also imagine the headaches of remembering which drive had what on it if they weren't organized. This dilemma would ideally be solved by arranging them in a logical heirarchical order.

First you would divide everything up into files that do work and files that are worked on - executable files and data files. If you have a great many executable files you might have to divide them into 2 or more sets. The data files must be divided into many more sets and subsets. There are data files like messages, there are tables, lists, subroutines, logs, graphics, music, controls, etc. Some files even defy catagorizing!

Which drives would get which files? Remember, we're talking about much power and large numbers of files! And, to make it interesting most of the commands you would use like 'deldir' aren't usually in memory, but on disk! To get all the power, THERE ARE SO MANY commands they'd choke the computer if you put them all in memory at the same time! And you'd go crazy looking for the right one on the disk - except for the organization.

(Pay close attention to the use of words like 'might' and 'could'. This is not a 'how to' article. I will go out of my way to avoid setting down 'how to' rules while hypothesizing.)

A level of organization is USUALLY, but not always (another variable) indicated by a '/' mark. Let's look at some levels.

Within the disk in a drive there might be a section of commands. Listed under commands (besides scores of things we usually think of as commands) might be WP, a word processor, and SS a spread sheet . Let's see what this starts to look like. Then I'll back up and make things better (and relieve those who just KNOW I'm doing this wrong.).

You start with a '/' for the top level /
You add a drive number /d0
You add another '/' for a level within d0 /d0/
You add the commands identifier /d0/cmds
You add another level indicator /d0/cmds/
Finally the name of the word processor /d0/cmds/wp

In BASIC you enter RUN"WP". In OS9 you enter /d0/cmds/wp. A 3 level command. (Nothing magic about the '3'. There can be more levels.) Let's do a tiny bit of computing - deleting WP from the disk. That will add another dimension to remember. If there were only one level you could enter 'del wp.' But the computer wants something like this: del /d0/cmds/wp The real beginners OS9 system usually starts with 2 drives at the top level of organization. So let's put a backup copy of WP on the other drive - drive 1. Let's see how complicated this can turn out:

copy /d0/cmds/wp /d1/cmds/wp

Already you can see something that is similar to DECB - spaces separate items within a statement. Something different is that both the word 'copy' and the word 'WP' are executable and considered commands on the disk. Some people separate these kinds of files making yet another thing to remember. Now remember that you have tons of stuff on your disk - commands

and data. The authors of OS9 came up with a way to cut down on a lot of the typing you need to do to slice through all the levels of organization. You can set the system to know what drive and/or level you're working with and whether it is executable or data. Voila, the commands CHD (data) and CHX (executable). (Let's switch to computer talk for these levels of organization.

What they are called is directories and sub-directories. Like a general index for an encyclopedia then a different index for each book.) If you enter "chx /d0/cmds" you set the executable commands pointer to that drive and directory. Then you can leave off that part from a typed command! Since WP is in the cmds directory, to get it you just enter "wp"! At this point you could delete it by entering "del wp." Simplifies things, doesn't it?

(Let's do some more computerese. After the command itself, a line like we've been using is called a pathlist. It indicates the path down through the levels of directories to a given item you're after. Remember - the effective name [pathlist] of a file includes the names of the drives and directories above the filename itself.) These helpful commands (CHD and CHX) are perhaps even more time saving if you're dealing with data. For example, if you have to decompress several files that are located in the ARC directory, you could enter:

"chd /d1/arc".

Then you could call all the files without the full command line and all the files you decompress (dearchive) would be stored in that same directory. (Still more. When you set chx to a directory it is then called the "execution" directory and the system will look there for any command you use that doesn't have a pathlist preceding it.) CHD and CHX are a lot like connecting to two single directory drives, one for data and one for commands.

That simplifies things doesn't it? But are you thinking what I'm thinking? Chd and chx are also two more things to remember! So what happens if you forget? You get an error!! If you set "chx /d1/cmds" and then call for a command that is on drive 0 you will get an "error 216" - pathname not found. That is like an NE error in DECB. But you can't just blame things on CHD or CHX, because the system will have to look in SOME specific place for what you command. And if you ask the wrong thing, you get the wrong thing.

Note that the error message uses "pathname", not "pathlist". Consider a pathname a segment (between slashes) of a pathlist. Let's look closer at errors. Forwarned is indeed forarmed. Usually errors are the bane of the OS9 beginners existance. With so many things to remember it is easy to make multiple mistakes. And as with other systems the actual error number doesn't always tell you what's wrong - even if you look up the number. But there is something you can keep in mind that will ease your recovery from such things with the least frustration and time.

Knowing it in advance will at least allow you to analyze problems in an intelligent way. Remember that the file name is called "pathlist" because it includes the path of drives or directories through which the file is accessed. Any part of the pathlist can give you the error - not just the file name at the end of it. Let me describe just one error problem. If the command you start a line with is not in the directory of executable commands that you indicate, you can get a 216 error there. If the drive you name, has the wrong disk in it, you can get an error from that part of the pathlist. If the directory you name is not on the indicated disk, that will give the same error, 216 - pathname not found. And, of course, if the file you want turns out not to be in the indicated directory, same thing.

Of course, misspelling can effect any of these things. At least you now know to look in different places for trouble. All this really involves a state of mind. To get at all that power you have to think differently (and more) with OS9 than you did in DECB. It's like moving into a house with 10 times as many rooms and 10 times the stuff. At first you'll have trouble remembering where you put everything.

You will find, as you progress, that there are alternative ways to do a given task. The third section will involve a little less attitude and a little more technique. There I will go a little deeper into some points I have made so far. Now section two. Beyond the System A sort of philosophy is also involved when you begin to deal with OS9 software. If you used only smoothly functioning, shrink wrapped commercial software, you probably wouldn't be reading this. You've probably been on Delphi to try the OS9 SIG's software or ask for help with something that is not so smooth running. There are beginners who frequent Delphi. Often they're as full of questions as you are, but they have answers. At the other end of the spectrum there are some very expert and brilliant programmers who know "everything there is to know" about OS9.

Too often, you will find yourself simply talking a different language than these "tech types." It isn't that they don't want to help. They're helping each other every day! If you want to know something about

the exact syntax of an obscure part of a brand new update to the latest XYZ language enhancement, you're in luck. But if you want to know how to get the menu on yesterdays spreadsheet it might take longer. Some very bright programmers dash off experimental programs for fun. Sometimes they like what they end up with and post it on Delphi.

Maybe they wrote 2 pages of documentations for a complicated communications program. Maybe they didn't finish it. Maybe they forgot they even wrote it! Grabbing the first thing in sight may or may not be a good idea. If you're looking to OS9 to be the cure for the common cold, the end all, be all, you need to rethink things. It is a tough thing to learn. But some of those brilliant programmers I mentioned have worked long and hard at getting rid of bugs in the original package and adding yet more power to it.

I guess the irony is that this system sits in such a tiny box wishing it had some place to go. Users require more and more function. That calls for more and bigger software, and that calls for more memory and/or data storage space. When that space is limited, you cut something. What should get cut?

A very appealing aspect of OS9 is the ability to snap from one program to another with the push of a button. But with space and CPU speed limited, how many programs of what size, power and function can you squeeze into a little CoCo? Working Delphi is nice. You run a terminal program. Maybe you have an editor to generate messages which are stored. You download a file to disk or ramdisk. Maybe you print out documentation in the background. Fun!

But what are you downloading? A spreadsheet that will hold all your small business bookkeeping for a year? Will you run that at the same time you have a memory gobbling graphics program in place? And the graphics user interface you use? And the big game you didn't finish last time you played?

Then, perhaps more importantly, there is the 6809 CPU a terrific little chip that could blow the doors off everything when it first came out (and later). DECB users who've been around a while remember when everyone was scrambling to get things running at double speed - (still less than 2 mhz). If one program can need that, what about several, running at once? Some programs take little CPU time. Some take a lot more, like when there is much disk I\O. Special disk controllers can help a lot, but that is only one of the angles.

How successful multi-tasking will be for you will depend on what you'll be running. Don't plan on having three action games running on three different terminals.

With applications growing to accomodate users demands for computing power, more memory and speed is needed. If you're a purist of the 68xx genre, the new 68xxx computers may be interesting. They have many times more memory and speed than CoCo. You'll find programmers right on Delphi who are working day and night to develop applications for these computers.

Take a look at things. If you're a budding programmer, learning OS9 can be very educational. If you're a hobbiest you can do a lot of experimenting with OS9 software and utilities. You can even run some of those very good applications programs. Or power a rock concert via Ultimuse and MIDI!

A little Body For the Attitude

So far, I've given you just a peek at how life can be in the OS9 world. Some things I've described may be very unlike what you ultimately find to be your specific experience. But I have been more interested in the general idea of things. Now I'll touch on a few points that will probably effect everyday computing more. Still, this isn't an OS9 "how to", just a whiff of its perfume.

Why is it so important to have so many levels of directories? Firstly, as I mentioned, there is a lot of stuff to organize. With people writing constantly there are new "tools" coming out almost every week! (I have 16 clocks on file and that isn't all of them.) By "tools" I mean utilities to do jobs, etc. Earlier I used "del" in an illustration. That is a very simple tool. "List" is a fairly simple tool. "Free" is like free in DECB. "Mfree" is like "MEM". (There ARE differences between such similar DECB and OS9 commands but the general idea is the same). Improved versions of various commands are written, often somewhat larger. Most veteran OS9 users have an entirely new shell - SHELL +.

Other commands are replaced if the user prefers something different (and if an alternative is available). But you really can't just toss out the original commands. There WILL be a time when you need at least some of them if you stick with OS9 long! So you keep copies of them. Probably on a separate disk. But if you happen to want to use both versions you might want them on one disk. You can't do that with DECB if you use the same name, but it is easy with OS9, because you can put them in different directories.

You might want to put them on the same level in

differently named directories. You could have, for example, CMDS1 and CMDS2. Then it would be possible to have one version of (perhaps) ATTR in one and another version in the other - (/d0/cmds1/attr or /d0/cmds2/attr). How much of this you can do might depend on your own memory.

Or you might want to put them on different levels. You could actually have a CMDS directory listed under the regular CMDS directory (/d0/cmds/cmds/attr), although it would probably not be a very good idea. A 'dir' could confuse you if you didn't remember which level you were looking at.

Another reason for various directories is the wide variety of data you can (sometimes must) have on disk. Frequently you will see directories named SYS, DOC, MODULES or ARC. SYS might have data essential to the function of an assembler. It might have 'help' or 'error' messages. 'DOC' would probably have documentation for programs or commands. 'MODULES' would likely have descriptions of the parameters of your various I/O devices such as your printer or RS-232 pak (40 isn't unusual). ARC would probably have archived (compressed) copies of programs, data, or other files.

As mentioned, directories CAN be 'stacked' many levels deep. One communications program stores dialing information like this: /d0/sys/dial/filename. A MultiVue screen icon might be named like this: /d0/cmds/icon/icon.app. I saw an example in which Basic09 commands were on the same disk with others. Under the regular CMDS directory was BASIC09, and under that was another CMDS. To copy it you could end up with this line:

```
copy /d0/Basic09/filename /d1/basic09/filename.
```

(Note, above, that I have put directory names in capital letters but the command lines in lowercase. If you have many things on the same level, caps make a directory stand out on the screen. When you set up a directory (using MAKDIR) you can just enter the name in caps. After that you need not use caps to call it. In most cases, OS9 accepts lowercase keyboard input for the system. Programs run under OS9 make their own decision on that).

If you see the reason for multiple directories, we can go on. At some point you will probably come across the phrase 'Unified I/O System'. So far I've referred to pathlists only as long paths through the maze of directories to files on your disks. But paths can access other things. Via pathlists you can list a text file to the screen or you can list it to your printer or to a screen in another window to be seen later! Via a pathlist you can read input from your RS-232 pak (and therefore whatever it is connected to). You can run a program in a window you don't see. These things can happen because of this 'unified' system.

In section one I mentioned that most commands are usually kept on disk. When you need, for example, the command 'format' it will probably be on disk. When you enter 'format' the computer loads it into memory and executes it (but not by firing squad) then unloads it when it is done with it. But you can also load commands into memory manually, then unload them when you wish.

(Computereze for 'unload' is 'unlink'.)

In section one I mentioned that mis-spelling something in a command line could cause an error such as 216 (pathname not found). Something that you can do in OS9 does make life easier when this and other things happen - the command line buffer. What you enter as a command line is saved in memory. You can then repeat it with the appropriate keypresses. As it comes, OS9 has an command repeat that is activated by CTRL-A. Before you press <ENTER> you can do some overstrike type editing.

With one very popular new version of this you use right arrow or shifted right arrow instead of CTRL-A. You can easily key over to the offending characters, change, insert or delete them and repeat the command. This is the 'SCF EDITOR' I mentioned at the beginning of this article.

This saves MUCH time over typing in whole lines again, especially when dealing with long full pathlists. If you're not the best typist, it can be a life saver. It also comes in handy if, for example, you have several similar files to copy. By changing only the actual filename part of the pathlist you could move several files without retyping much.

On the other hand, when in doubt about where you are, you can usually enter a full pathlist to 'feed the hungry computer'. (Sometimes, specific programs might not allow that).

There are other things that can really frustrate the beginner. One thing is the time it takes to do certain tasks according to the manual. For example, at some point you will wish to rewrite your startup file. Beginners haul out the 'build' command and start a new startup file from scratch as instructed by the manual. One tiny error and you're back to deleting, renaming, rewriting or what all, from the top down.

Startup is actually a text file called a script. (This may be in the manual somewhere, but I've never found it. It is definitely not in my indexes). Each command is on a separate line. If you can get hold of a line editor such as SLED from Delphi, (or a word processor) you can load such a script as "startup" and add or subtract at will. Just loading one and looking at it will take much of the mystery out of it.

Always press <ENTER> after each command as you would on the keyboard. Just as in regular word processing, this will start a new line for the next command.

If you wish to keep a copy of the original startup file you can rename it the way the manual suggests - before starting on a new one. Then when you are finished with the new one you save it to disk as "startup" just like the original. If your line editor has an overwrite feature (such as the one in SLED) you can just write the file into the space where the original was. Once you get comfortable with the system you will probably do this.

The "edit" utility that comes with OS9 can do this job. If you find it easy to use, by all means do so. But with a regular text editor you can see it all at once and use the arrow keys to edit.

Since a script is just a series of commands in a text file, you can write different scripts to do different things and call them just as you would any other command. This is a good way to set up a series of commands you use repeatedly. If a particular program calls for doing several things to prepare for it, you can probably do them in a script. You could put your preparatory commands in it, save it in the "cmds" directory as "pstart", for program start, and do the job by just entering "pstart" (assuming your chx was set to "cmds").

Scripts are really very easy to write. You'll see. But all is not rose petals here either. Sometimes, even when the script lines appear to be perfect, a line will just not work. Swapping lines sometimes makes a difference. Script lines do not give error messages. Shell+ will automatically get a script from the execution directory (set by chx). You can NOT load a script into memory like other commands unless you upgrade to Shell+ v2.0 or higher.

I have tried to say (perhaps warn), in a simple way, how complex OS9 can be. (Even the explanation has become somewhat complex.) I have tried to offer some preparation for dealing with that by covering an area or two that has been especially problematic for the new OS9 user - the manuals can be very frustrating sometimes. Perhaps the best preparation is to tell you that this is just the "tip of the iceberg."

The Firing Line

Since directories and the CHD and CHX commands are at the center of a lot of beginners problems here is still more to help make things clear.

When you type in a line like:

list /d1/modules/bootlist

the system looks for those names on the disk. It looks first for "list" in the execution directory then looks for a descriptor named "d0", a directory named "modules" and a file named "bootlist". It must have the numbers that these names represent, and it scours one or more disks looking for them in logical ways.

But when you use CHD and CHX those numbers are stored for quick use. If the "CMDS" directory is, for example, on line 1 to drive 1 and on sector 872, the CHX command stores that information. It is as if a big arrow were pointing to that sector. After that, any time the system "sees" a command it jumps to those numbers - that sector. That is why it cannot look for the "CMDS" directory by name.

Here are some typical command line problems.

Say you have put a system disk in drive 0 and issued a "CHX /d0/cmds" command. Then you enter:

copy /d0/sys/stdptrs /d1/sys/stdptrs

Things will probably be okay. But if you were in a hurry it may be:

cpy /d0/sys/stdptrs /d1/sys/strptrs

The system cannot find "cpy" and you get a 216 error.

If you put the disk in but forgot the CHX command, a perfect command line will probably give you a 215 error - bad pathname - because the system jumped to sector so-and-so and there was no "CMDS" directory there! See?? Right there is where a lot of frustration comes in. (Sometimes this may work, because if the command is in memory [where the system first looks] it will work anyway.)

"copy /d0/sys/stdptrs /d1/sys.stdptrs" isn't a hard mistake to make if you're not a good typist. This will cause the system to store a file in the directory last indicated by CHD and name it "sys.stdptrs."

'copy /d0/sys/strptrs /d0/sys/stdptrs' will probably give you a 218 error - file already exists.

'copy /d0/sys/stdptrs /d1/sys/stdprts' will not give you an immediate error. But when something looks for 'stdptrs' and there is nothing but 'strprts' it is 216 time again.

Believe it or not, if you have set CHX to /d1/cmds, for instance, and you decide to use a command (let's say 'list') which is in your /d0/cmds directory you can enter something like:

/d0/cmds/list /d0/txt/message.

Lastly let me touch on redirection again. While making up a new bootfile (OS9Boot) I had some problems. I decided to use the 'ident' utility to look at what was in the file. With 30 or 40 or more modules in OS9Boot you just can't get the idea on the screen. So I redirected the output of the ident. A normal ident on a single module might look like this:

ident /d0/modules/cc3disk.dr

By redirecting output like this:

ident /d0/OS9Boot >/p

I had all of the modules information sets printed out in hardcopy instead of the screen.

You don't have to be typing command lines to get errors. Some utilities REQUIRE certain CHD settings, and if you just forgot to do one the utility could be getting the wrong information or none at all.

To really bring your system up to date, RICKULAND is distributing some very cheap sets of all the things you need to make a new system out of what you started out with. Contact him on the OS9 Sig.

This is really a good idea, because you can spend days, weeks or even months acquiring all the patches and getting them flawlessly into place. AND you will find that some of the very good public domain software available does not work well unless the original bugs are eliminated from the operating system itself.

I'm going to mention something Rick suggested, so you can 'chew' on it. It comes with a sort of catch 22 built in so be careful how you size it up.

Although the OS9Boot file must be in one contiguous file, not stored in various parts of the disk, it does not need to start at the beginning of the disk. So you could put your commands directory on the disk first - on all your system disks - and the pointer set up by the CHX command would be the same for every disk. Therefore you would not have to reset it except for times when you used /d1.

The trick here is that by the time you might feel you have the expertise to do this you would probably have several system disks set up the other way. Those disks would require 'CHXing' as usual and the mix might not be worth the effort.

Jim LaLone (upper middle class neophyte) On Delphi: TERMITE

oooooooooo0000000000oooooooooo

FROM_THE_LIBRARIAN

Have you come to the same problem I came to lately? After updating my Boot file several times with modules from the public domain library, I could not tell later where these were coming from. When "Ident" gives you edition and revision numbers, those might not have been changed by the programmer who wrote the new version. And what about those you patched with "Ipatch"? I found that the only reliable information is the CRC number, and if I had a listing of all the versions I know of a particular module with their CRC numbers, life would be much easier. So, here are a few:

Module	CRC	Origin	Comments
cc3io	F737C2	OS9 Level II System Disk	cc3io.dr
cc3io	923705	Public Domain Library #11	serialmouse/cc3io.joy.ipc
rel	6FD34C	OS9 Level II System Disk	
rel	B1F86C	Public Domain Library #11	krnl.ipc
boot	03DC4E	OS9 Level II System Disk	
boot	8D0496	Public Domain Library #11	krnl.ipc
os9p1	969A94	OS9 Level II System Disk	
os9p1	C21516	Public Domain LIbrary #11	krnl.ipc
r0	04B49F	Kevin Darling	Rammer.ar
r0	12523C	OS9 L.II Development System	r0_96k.dd
r0	A42C1D	OS9 L.II Development System	r0_128k.dd
r0	30083C	OS9 L.II Development System	r0_192k.dd
ram	E863B3	OS9 L.II Development System	ram.dr
rammer	15C571	Kevin Darling	Rammer.ar

They will be more for you next time. Happy computing!
Jean-Pierre